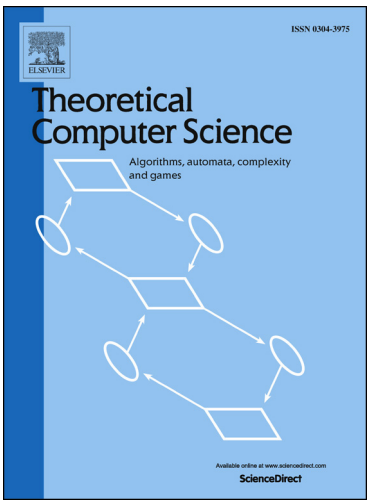
Accepted Manuscript



A logic for the stepwise development of reactive systems

Alexandre Madeira, Luis S. Barbosa, Rolf Hennicker, Manuel A. Martins

PII: S0304-3975(18)30143-9
DOI: https://doi.org/10.1016/j.tcs.2018.03.004
Reference: TCS 11504

To appear in: *Theoretical Computer Science*

Received date: 13 April 2017
Revised date: 14 November 2017
Accepted date: 1 March 2018

Please cite this article in press as: A. Madeira et al., A logic for the stepwise development of reactive systems, *Theoret. Comput. Sci.* (2018), https://doi.org/10.1016/j.tcs.2018.03.004

A logic for the stepwise development of reactive systems

Alexandre Madeira[a,c], Luis S. Barbosa[a], Rolf Hennicker[b], Manuel A. Martins[c]

[a] *QuantaLab and HASLab INESC TEC, Universidade do Minho, Portugal*
[b] *Ludwig-Maximilians-Universität München, Germany*
[c] *CIDMA - Dep. Mathematics, Univ. Aveiro, Portugal*

Abstract

$\mathcal{D}^{\downarrow}$ is a new dynamic logic combining regular modalities with the binder constructor typical of hybrid logic, which provides a smooth framework for the stepwise development of reactive systems. Actually, the logic is able to capture system properties at different levels of abstraction, from high-level safety and liveness requirements, to constructive specifications representing concrete processes. The paper discusses its semantics, given in terms of reachable transition systems with initial states, its expressive power and a proof system. The methodological framework is in debt to the landmark work of D. Sannella and A. Tarlecki, instantiating the generic concepts of constructor and abstractor implementations by standard operators on reactive components, e.g. relabelling and parallel composition, as constructors, and bisimulation for abstraction.

Keywords: Specification, Reactive systems, Dynamic logic, Hybrid logic

1. Introduction

Almost 30 years ago, D. Sannella and A. Tarlecki claimed, in what would become a most influential paper in (formal) Software Engineering [31], that *"the program development process is a sequence of implementation steps leading from a specification to a program"*. Being rather vague on what was to be understood either by specifications (*"just finite syntactic objects of some kind"* which *"describe a certain signature and a class of models over it"*) or programs (*"which for us are just very tight specifications"*), the paper focuses entirely on the development process, based on a notion of refinement.

Indeed, the quest for suitable notions of *implementation* and *refinement* has been for more than four decades on the research agenda for rigorous Software Engineering. This goes back to Hoare's paper on data refinement [19], which influenced the whole family of model-oriented methods, starting with VDM [21]. A recent reference [33] collects a number of interesting refinement case studies in the B method, probably the most successful member of the family in what concerns industrial applications.

In such model-oriented approaches, a specification is said to refine another one if every model of the latter is a model of the former. Sannella and Tarlecki's work complemented and generalised this view with the notions of *"constructor"* and *"abstractor implementations"*:

> *"**constructor** implementations which involve a construction 'on top of' the **implementing** specification, and **abstractor** implementations which additionally provide for abstraction from some details of the **implemented** specification"* [31].

The idea behind a constructor implementation is that for representing a specification *SP* one may use one or several given specifications and apply a construction on top of them to satisfy the requirements of *SP*. On the other hand, sbstractor implementations capture the fact that sometimes the requirements for a system are only satisfied up to an abstraction which usually involves hiding of implementation details. Over time, many others contributed along similar paths, with Sannella and Tarlecki's specific view later consolidated in their landmark book [32]. All main ingredients were already there: i) the emphasis on *loose* specifications; ii) correctness by construction, guaranteed by vertical compositionality, and iii) genericity, as the development process is independent, or parametric, on whatever logical system better captures the requirements to be handled.

The present article investigates this approach in the context of reactive software, *i.e.* systems which interact with their environment along the whole computation, and not only in its starting and termination points [1]. The relevance of such an effort is anticipated in Sannella and Tarlecki's book [32] itself: *"An example of an area for which a satisfactory, commonly accepted solution still seems to be outstanding (despite numerous proposals and active research) is the theory of concurrency"* (page 157). Different approaches in that direction have been proposed, of which we single out an extension to concurrency in K. Havelund's PhD thesis [17]. His work, however, focused essentially on functional requirements expressed by algebraic specifications and implemented in a functional programming language.

As a matter of fact, the development of reactive systems, which are nowadays the norm rather than the exception, followed a different path. Typical approaches

start from the construction of a concrete model (e.g. in the form of a transition system [34], a Petri net [29] or a process algebra expression [20, 4]) upon which the relevant properties are later formulated in a suitable (modal) logic and typically verified by some form of model-checking. Resorting to old software engineering jargon, most of these approaches proceed by *inventing & verifying*, whereas this paper takes the alternative *correct by construction* perspective.

Actually, our research hypothesis is that also in the domain of reactive systems, loose specification has an important role to play, because it supports the gradual incorporation of further requirements and implementation decisions such that verification of the correctness of a complex system can be done piecewise in smaller steps. Additionally, this allows for the systematic documentation of design decisions, as a support to systems' maintenance and refactoring.

Therefore, the challenge undertaken here is twofold. First, we propose a new logic to support the development of reactive systems at different levels of abstraction. Then, we show how to adapt to this context Sannella and Tarlecki's recipe according to which *"specific notions of implementation (...) corresponds to a restriction on the choice of constructors and abstractors which may be used"* [31].

To address these challenges, we introduce a new logic, $\mathcal{D}^{\downarrow}$, which is able not only to express abstract properties, such as liveness requirements or deadlock avoidance, but also to describe the concrete, recursive process structures which implement them. The logic combines modalities indexed by regular expressions of actions, as in dynamic logic [16], and state variables and binders, characteristic of hybrid logic [7].

As a second contribution, the paper introduces a number of constructors and abstractors relevant to the development of reactive systems. Interestingly, it turns out that requirements of Sannella and Tarlecki's methodology for vertical composition of abstractor/constructor implementations boils down to the congruence property of bisimilarity w.r.t. constructions on labelled transition systems, like parallel composition and relabelling.

This article is an extended version of our previous work [24], presented at ICTAC'2016. As such it includes the complete proofs of all results, and two new sections: Section 5 discusses the expressive power of $\mathcal{D}^{\downarrow}$, while Section 6 introduces a sound proof calculus for it.

Apart from those new sections, section 2 introduces $\mathcal{D}^{\downarrow}$, and sections 3 and 4, respectively, characterise the development method, with a brief revision of the relevant background, and its tuning to the design of reactive systems. Finally, section 7 concludes and points out some issues for future work.

2. A Dynamic Logic with Binders

2.1. $\mathcal{D}^{\downarrow}$: Syntax and Semantics

$\mathcal{D}^{\downarrow}$ logic is designed to express properties of reactive systems, from abstract safety and liveness requirements, down to concrete design decisions specifying the (recursive) structure of processes. It thus combines modalities with regular expressions, as originally introduced in dynamic logic [16], and binders in state variables. This logic retains from hybrid logic [7], only state variables and the binder operator first studied by V. Goranko in [13]. These motivations are reflected in its semantics. Differently from what is usual in modal logics, whose semantics is given by Kripke structures and satisfaction evaluated globally in each model, $\mathcal{D}^{\downarrow}$ models are reachable transition systems with initial states at which satisfaction is evaluated.

Definition 1 (Model). For a finite set of atomic actions A, *models* are reachable A-labelled transition systems, i.e. triples (W, w_0, R) where W is a *set of states*, $w_0 \in W$ is the *initial state* and $R = (R_a \subseteq W \times W)_{a \in A}$ is a family of *transition relations* such that, for each $w \in W$, there is a finite sequence of transitions $R_{a^k}(w^{k-1}, w^k)$, $1 \leq k \leq n$, with $w^k \in W$, $a^k \in A$, such that $w_0 = w^0$ and $w^n = w$.

The *set of (structured) actions*, $\mathrm{Act}(A)$, induced by A is given by

$$\alpha \ni a \mid \alpha; \alpha \mid \alpha + \alpha \mid \alpha^*$$

where $a \in A$.

Let X be an infinite set of variables, disjoint with A. A valuation for an A-model $\mathcal{M} = (W, w_0, R)$ is a function $g : X \to W$. Given such a g and $x \in X$, $g[x \mapsto w]$ denotes the valuation given by $g[x \mapsto w](x) = w$ and $g[x \mapsto w](y) = g(y)$ for any other $y \neq x \in X$.

Definition 2 (Formulas and sentences). The set $\mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A)$ of *A-formulas* is given by

$$\varphi ::= \mathbf{tt} \mid \mathbf{ff} \mid x \mid \downarrow x.\varphi \mid @_x\varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

where $x \in X$ and $\alpha \in \mathrm{Act}(A)$. $\mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A) = \{\varphi \in \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A) | \mathrm{FVar}(\varphi) = \emptyset\}$ is the set of A-sentences, where $\mathrm{FVar}(\varphi)$ are the free variables of φ, defined as usual with $\downarrow$ being the unique operator binding variables.

$\mathcal{D}^{\downarrow}$ retains from hybrid logic the use of binders, but omits nominals: only state variables are used, even as parameters to the satisfaction operator ($@_x$). By doing so, the logic becomes restricted to express properties of states reachable from the initial state, i.e. processes.

To define the satisfaction relation we need to clarify how composed actions are interpreted in models. Let $\alpha \in \mathrm{Act}(A)$ and $\mathcal{M} \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$. The interpretation of an action α in $\mathcal{M}$ extends the interpretation of atomic actions by $R_{\alpha;\alpha'} = R_\alpha \cdot R_{\alpha'}$, $R_{\alpha+\alpha'} = R_\alpha \cup R_{\alpha'}$ and $R_{\alpha^*} = (R_\alpha)^\star$, with the operations $\cdot$, $\cup$ and $\star$ standing for relational composition[1], union and Kleene closure.

Given an A-model $\mathcal{M} = (W, w_0, R)$, $w \in W$ and $g : X \to W$,

- $\mathcal{M}, g, w \models \mathbf{tt}$ is true; $\mathcal{M}, g, w \models \mathbf{ff}$ is false;
- $\mathcal{M}, g, w \models x$ iff $g(x) = w$;
- $\mathcal{M}, g, w \models{\downarrow}\, x.\, \varphi$ iff $\mathcal{M}, g[x \mapsto w], w \models \varphi$;
- $\mathcal{M}, g, w \models @_x\varphi$ iff $\mathcal{M}, g, g(x) \models \varphi$;
- $\mathcal{M}, g, w \models \langle\alpha\rangle\varphi$ iff there is a $w' \in W$ with $(w, w') \in R_\alpha$ and $\mathcal{M}, g, w' \models \varphi$;
- $\mathcal{M}, g, w \models [\alpha]\varphi$ iff for any $w' \in W$ with $(w, w') \in R_\alpha$ it holds $\mathcal{M}, g, w' \models \varphi$;
- $\mathcal{M}, g, w \models \neg\varphi$ iff it is false that $\mathcal{M}, g, w \models \varphi$;
- $\mathcal{M}, g, w \models \varphi \wedge \varphi'$ iff $\mathcal{M}, g, w \models \varphi$ and $\mathcal{M}, g, w \models \varphi'$;
- $\mathcal{M}, g, w \models \varphi \vee \varphi'$ iff $\mathcal{M}, g, w \models \varphi$ or $\mathcal{M}, g, w \models \varphi'$.

We write $\mathcal{M}, w \models \varphi$ if, for any valuation $g : X \to W$, $\mathcal{M}, g, w \models \varphi$. If φ is a sentence, then the valuation is irrelevant, i.e., $\mathcal{M}, g, w \models \varphi$ iff $\mathcal{M}, w \models \varphi$. For each sentence $\varphi \in \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A)$, we write $\mathcal{M} \models \varphi$ whenever $\mathcal{M}, w_0 \models \varphi$. Observe again the pertinence of avoiding nominals: if a formula is satisfied in the standard semantics of hybrid logic, then it is satisfiable in $\mathcal{D}^{\downarrow}$. Obviously, this would not happen in the presence of nominals.

The remaining of this section discusses the versatility of $\mathcal{D}^{\downarrow}$ claimed in the introductory section. In the sequel, given a set of atomic actions $A = \{a_1, \ldots, a_n\}$, we write A and $-a_i$ to refer to structured actions $a_1 + \ldots + a_n$, and and $a_1 + \ldots + a_{i-1} + a_{i+1} + \ldots + a_n$, respectively.

By borrowing regular modalities from dynamic logic [16, 15], $\mathcal{D}^{\downarrow}$ is able to express liveness requirements such as "*after the occurrence of an action a, an action b can be eventually realised*" with $[A^*; a]\langle A^*; b\rangle\mathbf{tt}$, or "*after the occurrence of an action a, an occurrence of an action b is eventually possible if it has not occurred before*" with $[A^*; a; (-b)^*]\langle A^*; b\rangle\mathbf{tt}$. Safety properties are also captured by sentences of the form $[A^*]\varphi$. In particular, *deadlock freeness* is expressed by $[A^*]\langle A\rangle\mathbf{tt}$.

[1]Symbol $\cdot$ (rather than the more standard ;) is used throughout the paper to denote diagrammatic composition of binary relations, to distinguish the sequential action composition from its semantic denotation.

Example 1. As a running example we consider a product line with a stepwise development of a file compressing service, working both with text and image files. We start with an abstract requirements specification SP_0, over the set $A = \{inTxt, inGif, outZip, outJpg\}$ of atomic actions. Informally, *inTxt* (respectively, *inGif*) stands for the input of a txt-file (respectively, a gif-file), and action *outZip* (respectively, *outJpg*) for the output of a zip-file (respectively, a jpg-file). Sentences (0.1)-(0.3) below express three requirements: (0.1) Whenever a txt-file has been received for compression, the next action must be an output of a zip-file, (0.2) whenever a gif-file has been received, the next action must be an output of a jpg-file, and (0.3) the system should never terminate.

(0.1) $[A^*; inTxt]\big(\langle outZip\rangle \mathbf{tt} \wedge [-outZip]\mathbf{ff}\big)$
(0.2) $[A^*; inGif]\big(\langle outJpg\rangle \mathbf{tt} \wedge [-outJpg]\mathbf{ff}\big)$
(0.3) $[A^*]\langle A\rangle \mathbf{tt}$

Obviously, SP_0 is a very loose specification of rudimentary requirements with a huge set of possible models. □

The logic $\mathcal{D}^{\downarrow}$, however, is also suited to directly express process structures and, thus, the implementation of abstract requirements. The binder operator is crucial for this. The ability to give names to visited states, together with the modal features to express transitions, makes possible a precise description of the whole dynamics of a process in a single sentence. Binders allow to express recursive patterns, namely loop transitions (from the current to some visited state). Actually, this kind of properties cannot be specified in the absence of a feature to refer to specific states in a model, as in standard modal logic. For example, sentence

$$\downarrow x_0.\big(\langle a\rangle x_0 \wedge \langle b\rangle \downarrow x_1.(\langle a\rangle x_0 \wedge \langle b\rangle x_1)\big) \tag{1}$$

specifies a process with two states accepting actions a and b respectively. As discussed in the sequel, the stepwise development of a reactive system typically leads to a set of requirements defining concrete transition systems. These are expressed in the fragment of $\mathcal{D}^{\downarrow}$ omitting modalities indexed by the Kleene closure of actions, that can be directly translated into a set of FSP [25] definitions. Fig. 1 depicts the translation of the formula above as computed by a proof-of-concept implementation of such a translator[2]. Note, however, that sentence (1) is a loose specification of the envisaged scenario (e.g. a single state system looping on a and b also satisfies this requirement). Resorting to full $\mathcal{D}^{\downarrow}$ concrete processes, unique up

[2]see `translator.nrc.pt`.

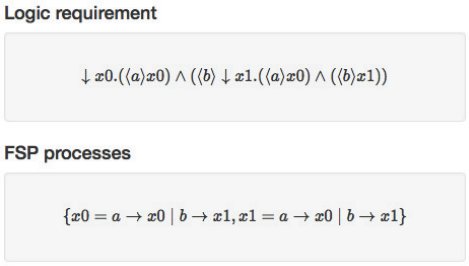


Figure 1: D2FSP Translator: Translating $\mathcal{D}^{\downarrow}$ into FSP processes.

to isomorphism, can be defined, i.e. we may introduce monomorphic specifications. For this specific example, it is enough to consider, in the conjunction guarded by x_1, the term $@_{x_1}\neg x_0$ (to distinguish between the states bound by x_0 and x_1), as well as to enforce determinism resorting to formula (det) in Ex. 3 below.

2.2. Turning $\mathcal{D}^{\downarrow}$ into an Institution

The concept of an *institution* has been introduced by Joseph Goguen and Rod Burstall in [11]. An institution formalises some basic ingredients that any logical system should provide when it is used as a specification framework in program development. The notion relies on a clear separation between *syntax* (signatures, sentences) and *semantics* (models) which are related by a satisfaction relation $M \models \varphi$ between models and sentences.

In order to meet the necessary requirements to adopt Sannella and Tarlecki's development method, logic $\mathcal{D}^{\downarrow}$ has to be framed as a logical institution [11].

In this view, our first concern is about the *category of signatures*. As suggested, signatures for $\mathcal{D}^{\downarrow}$ are finite sets A of *atomic actions*, and a signature morphism $A \xrightarrow{\sigma} A'$ is just a function $\sigma : A \to A'$. Clearly, this defines a category, $\mathrm{Sign}^{\mathcal{D}^{\downarrow}}$.

Our second concern is about the *models functor*. Given two models, $\mathcal{M} = (W, w_0, R)$ and $\mathcal{M}' = (W', w_0', R')$, for a signature A, a *model morphism* (A-morphism, for short)is a function $h : W \to W'$ such that $h(w_0) = w_0'$ and, for each $a \in A$, if $(w_1, w_2) \in R_a$ then $(h(w_1), h(w_2)) \in R_a'$. Clearly, the class of models for A, and the corresponding morphisms, defines a category $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$.

Definition 3 (Model reduct). Let $A \xrightarrow{\sigma} A'$ be a signature morphism and $\mathcal{M}' = (W', w_0', R')$ an A'-model. The *σ-reduct of $\mathcal{M}'$* is the A-model $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}') = (W, w_0, R)$ such that

- $w_0 = w_0'$;
- W is the largest set with $w_0' \in W$ and, for each $v \in W$, either $v = w_0'$ or there is a $w \in W$ such that $(w, v) \in R'_{\sigma(a)}$, for some $a \in A$;

- for each $a \in A$, $R_a = R'_{\sigma(a)} \cap W^2$.

Lemma 1. Consider a signature morphism $A \xrightarrow{\sigma} A'$, a A'-model $M' = (W', w'_0, R')$ and its σ-reduct $\mathcal{M} = (W, w_0, R)$. Then, for any action $\alpha \in \mathrm{Act}(A)$,

1. $R_\alpha = R'_{\overline{\sigma}(\alpha)} \cap W$, and
2. for any $w, v \in W'$ such that $(w, v) \in R'_{\overline{\sigma}(\alpha)}$, $w \in W$ iff $v \in W$.

Proof. The proof is by induction on the structure of actions. The property holds by definition for basic actions $a \in A$. We consider below the case of sequential composition of actions $(\alpha; \alpha')$; the remaining cases follow a similar argument.

$$
\begin{array}{ll}
 & R_{\alpha;\alpha'} \\
= & \quad \{\ ;\ \text{defn}\} \\
 & R_\alpha \cdot R_{\alpha'} \\
= & \quad \{\ \text{I.H.}\} \\
 & (R'_{\overline{\sigma}(\alpha)} \cap W^2) \cdot (R'_{\overline{\sigma}(\alpha)} \cap W^2)
\end{array}
$$

Hence,

$$
\begin{array}{ll}
 & (w, v) \in (R'_{\overline{\sigma}(\alpha)} \cap W^2) \cdot (R'_{\overline{\sigma}(\alpha)} \cap W^2) \\
\Leftrightarrow & \quad \{\ \cdot\ \text{defn}\} \\
 & (\exists z)\big((w, z) \in (R'_{\overline{\sigma}(\alpha)} \cap W^2) \wedge (z, v) \in (R'_{\overline{\sigma}(\alpha')} \cap W^2)\big) \\
\Rightarrow & \quad \{\ \text{set theory}\} \\
 & (\exists z)\big((w, z) \in (R'_{\overline{\sigma}(\alpha)} \wedge (z, v) \in R'_{\overline{\sigma}(\alpha')}\big) \wedge \\
 & (\exists z)\big((w, z) \in W^2 \wedge (z, v) \in W^2)\big) \\
\Leftrightarrow & \quad \{\ \cdot\ \text{defn}\ \} \\
 & (w, v) \in (R'_{\overline{\sigma}(\alpha)} \cdot R'_{\overline{\sigma}(\alpha')}) \cap (W^2 \cdot W^2) \\
\Rightarrow & \quad \{\ \cap\ \text{monotonicity (since } W^2 \cdot W^2 \subseteq W^2) + \overline{\sigma}\ \text{defn}\} \\
 & (w, v) \in (R'_{\overline{\sigma}(\alpha;\alpha')}) \cap W^2
\end{array}
$$

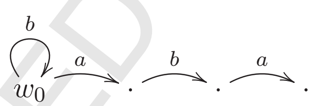
Therefore $R_{\alpha;\alpha'} \subseteq R'_{\overline{\sigma}(\alpha;\alpha')} \cap W^2$. For the converse direction:

$$
\begin{array}{ll}
 & R'_{\overline{\sigma}(\alpha;\alpha')} \cap W^2 \\
= & \qquad \{\ \overline{\sigma} \text{ and } ; \text{ defn}\} \\
 & (R'_{\overline{\sigma}(\alpha)} \cdot R'_{\overline{\sigma}(\alpha')}) \cap W^2 \\
\subseteq & \qquad \{\ \cdot \text{ monotonicity}\} \\
 & \big((R'_{\overline{\sigma}(\alpha)} \cap W^2) \cdot (R'_{\overline{\sigma}(\alpha')} \cap W^2)\big) \cap W^2 \\
= & \qquad \{\ \text{I.H.}\}
\end{array}
\quad\Bigg|\quad
\begin{array}{ll}
 & (R_\alpha \cdot R_{\alpha'}) \cap W^2 \\
= & \qquad \{\ R_\alpha, R_{\alpha'} \subseteq W^2\} \\
 & R_\alpha \cdot R_{\alpha'} \\
= & \qquad \{\ ; \text{ defn}\} \\
 & R_{\alpha;\alpha'}
\end{array}
$$

□

Moreover, given any A'-morphism $M'_1 \xrightarrow{h} M'_2$, it is easy to check that $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(M'_1) \xrightarrow{h} \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(M'_2)$ is also a (Nom, A)-morphism.

Example 2. Let us consider the inclusion signature morphism $\{a\} \xhookrightarrow{\sigma} \{a, b\}$ and an $\{a, b\}$-model $\mathcal{M}$ depicted by



The reduct of $\mathcal{M}$ is $w_0 \xrightarrow{a} \cdot$.

Lemma 2. For each A'-morphism $M'_1 \xrightarrow{h'} M'_2$ between two A'-models and for each signature morphism $A \xrightarrow{\sigma} A'$, we have a morphism $M_1 \xrightarrow{h} M_2$, where $M_1 = \mathrm{Mod}^{\mathcal{I}}(\sigma)(M'_1)$, $M_2 = \mathrm{Mod}^{\mathcal{I}}(\sigma)(M'_2)$ and h is the restriction of h' to W_1.

Proof. The proof is by induction over the structure of W_1. Note first that h is well defined in the sense that, for each $w \in W_1$, $h(w) \in W_2$:

- for the initial state $w_0^1 \in W_1$, we have by definition of h and since h' is a morphism, $h(w_0^1) = h'(w_0^1) = w_0^2$. By reduct definition, $w_0^2 \in W_2$.
- for each $v \in W_1$, there is a $w \in W_1$ such that $(w, v) \in R'^2_{\sigma(a)}$ for some $a \in A$. Since h' is a morphism, we have also that $(h(w), h(v)) \in R'^2_{\sigma(a)}$. Assume, by I.H., that $h(w) \in W_2$. Then, by reduct definition, $h(v) \in W_2$.

The morphism properties for h are directly inherited from the morphism properties of h'. □

Model morphisms are preserved by reducts, in the sense that, for each such morphism $h : \mathcal{M}'_1 \to \mathcal{M}'_2$ there is another $h' : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'_1) \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'_2)$, where h' is the restriction of h to the states of $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'_1)$. Hence, for each signature morphism $A \xrightarrow{\sigma} A'$, a functor $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma) : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A') \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$ maps models and morphisms to the corresponding reducts. Finally, this lifts to a contravariant *models functor*, $\mathrm{Mod}^{\mathcal{D}^{\downarrow}} : (\mathrm{Sign}^{\mathcal{D}^{\downarrow}})^{op} \to \mathbb{C}at$, mapping each signature to the category of its models and, each signature morphism to its reduct functor.

The third concern relates to the definition of the *functor of sentences*. Each signature morphism $A \xrightarrow{\sigma} A'$ can be extended to a formulas' translation $\hat{\sigma} : \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A) \to \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A')$ by identifying variables and replacing, symbol by symbol, each action by the respective σ-image. In particular, $\hat{\sigma}(\downarrow x.\varphi) = \downarrow x.\hat{\sigma}(\varphi)$ and $\hat{\sigma}(@_x\varphi) = @_x\hat{\sigma}(\varphi)$. Since $\mathrm{FVar}(\varphi) = \mathrm{FVar}(\hat{\sigma}(\varphi))$, for each signature morphism $A \xrightarrow{\sigma} A'$, we can define a translation of sentences $\mathrm{Sen}^{\mathcal{D}^{\downarrow}}(\sigma) : \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A) \to \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A')$, by $\mathrm{Sen}^{\mathcal{D}^{\downarrow}}(\sigma)(\varphi) = \hat{\sigma}(\varphi)$, $\varphi \in \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A)$. This defines the intended functor $\mathrm{Sen}^{\mathcal{D}^{\downarrow}} : \mathrm{Sign}^{\mathcal{D}^{\downarrow}} \to \mathbb{S}et$, mapping each signature to the set of its sentences, and each signature morphism to the corresponding translation of sentences.

Finally, our fourth concern is on the agreement of the satisfaction relation w.r.t. the *satisfaction condition*. This is established in the following result:

Theorem 1. Let $\sigma : A \to A'$ be a signature morphism, $\mathcal{M}' = (W', w'_0, R') \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A')$, $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}') = (W, w_0, R)$ and $\varphi \in \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A)$. Then, for any $w \in W (\subseteq W')$ and for any valuation $g : X \to W$ and $g' : X \to W'$, such that, $g(x) = g'(x)$ for all $x \in \mathrm{FVar}(\varphi)$, we have

$$\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, w \models \varphi \text{ iff } \mathcal{M}', g', w \models \hat{\sigma}(\varphi)$$

Proof. The proof is by induction on the structure of formulas. For that, we denote $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}')$ by (W, w_0, R). With exception of formulas x, $\langle\alpha\rangle\varphi$, $[\alpha]\varphi$ and $\downarrow x.\varphi$, the proof of all the cases is trivial. Moreover, the arguments for $\langle\alpha\rangle\varphi$ and for $[\alpha]\varphi$ are analogous. Hence, we only consider the proofs for the following cases:

Formulas x:

$$
\begin{aligned}
& \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, w \models x \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& w = g(x) \\
\Leftrightarrow \quad & \{ \text{ by hypothesis } g(x) = g'(x) \} \\
& w = g'(x) \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& \mathcal{M}', g', w \models x \\
\Leftrightarrow \quad & \{ \hat{\sigma} \text{ defn} \} \\
& \mathcal{M}', g', w \models \hat{\sigma}(x)
\end{aligned}
$$

Formulas $\downarrow x.\varphi$:

$$
\begin{aligned}
& \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, w \models \downarrow x.\varphi \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g[x \mapsto w], w \models \varphi \\
\Leftrightarrow \quad & \{ \text{ step } (\star), \text{ and I.H.} \} \\
& \mathcal{M}', g'[x \mapsto w], w \models \hat{\sigma}(\varphi), \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& \mathcal{M}', g', w \models \downarrow x.\hat{\sigma}(\varphi) \\
\Leftrightarrow \quad & \{ \hat{\sigma} \text{ defn} \} \\
& \mathcal{M}', g', w \models \hat{\sigma}(\downarrow x.\varphi)
\end{aligned}
$$

The step marked with a $(\star)$ is justified as follows: By hypothesis $g(y) = g'(y)$, for any $y \in \mathrm{FVar}(\downarrow x.\varphi)$. Hence, $g[x \mapsto w](y) = g'[x \mapsto w](y)$, for any $y \in \mathrm{FVar}(\varphi)$, and the induction hypothesis apply.

Formulas $@_x\varphi$:

$$
\begin{aligned}
& \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, w \models @_x\varphi \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, g(x) \models \varphi \\
\Leftrightarrow \quad & \{ \ g(x) = g'(x), \text{ and I.H.} \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathcal{M}', g', g'(x) \models \hat{\sigma}(\varphi), \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& \mathcal{M}', g', w \models @_x \hat{\sigma}(\varphi) \\
\Leftrightarrow \quad & \{ \hat{\sigma} \text{ defn} \} \\
& \mathcal{M}', g', w \models \hat{\sigma}(@_x \varphi)
\end{aligned}
$$

Formulas $\langle\alpha\rangle\varphi$:

$$
\begin{aligned}
& \text{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, w \models \langle\alpha\rangle\varphi \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& \text{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, v \models \varphi \text{ for some } v \in W \\
& \text{such that } (w, v) \in R_\alpha \\
\Leftrightarrow \quad & \{ \text{Lemma 1, and I.H.} \} \\
& \mathcal{M}', g', v \models \hat{\sigma}(\varphi) \text{ for some } v \in W' \\
& \text{such that } (w, v) \in R'_{\overline{\sigma}(\alpha)} \\
\Leftrightarrow \quad & \{ \models \text{defn} \} \\
& \mathcal{M}', g', w \models \langle\overline{\sigma}(\alpha)\rangle\hat{\sigma}(\varphi) \\
\Leftrightarrow \quad & \{ \hat{\sigma} \text{ defn} \} \\
& \mathcal{M}', g', w \models \hat{\sigma}(\langle\alpha\rangle\varphi)
\end{aligned}
$$

□

In particular:

Theorem 2 (Satisfaction condition). For any signature morphism $A \xrightarrow{\sigma} A' \in \text{Sign}^{\mathcal{D}^{\downarrow}}$, model $\mathcal{M}' \in \text{Mod}^{\mathcal{D}^{\downarrow}}(A')$ and sentence $\varphi \in \text{Sen}^{\mathcal{D}^{\downarrow}}(A)$,

$$\text{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}') \models \varphi \text{ iff } \mathcal{M}' \models \text{Sen}^{\mathcal{D}^{\downarrow}}(\sigma)(\varphi).$$

Proof. Since $\varphi \in \text{Sen}^{\mathcal{D}^{\downarrow}}(A)$, we have $\text{FVar}(\varphi) = \emptyset$, and hence, by Lemma 1, for any $w \in W$,

$$\text{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), w \models \varphi \text{ iff } \mathcal{M}', w \models \text{Sen}^{\mathcal{D}^{\downarrow}}(\sigma)(\varphi).$$

Moreover, by reduct definition, $w_0 = w'_0 \in W$, and the result follows. □

3. Formal Development *à la* Sannella & Tarlecki

Developing correct programs from specifications entails the need for a suitable logic setting in which meaning can be assigned both to specifications and their refinements. Sannella and Tarlecki have proposed a formal development methodology [31, 32] which is presented in a generic way for arbitrary institutions. As already pointed out in the Introduction, Sannella and Tarlecki have studied various algebraic institutions to illustrate their methodology and they presume the lack of a satisfactory solution in the theory of concurrency. In this section we briefly summarize their crucial principles for formal program development over an arbitrary institution, and illustrate the case of simple implementations by examples of our $\mathcal{D}^{\downarrow}$-logic institution. The concepts of constructor and abstractor implementations will be instantiated for $\mathcal{D}^{\downarrow}$ later on in Sect. 4.

In the sequel we assume given an arbitrary institution, with category Sign of signatures and signature morphisms, sentences functor Sen : Sign $\to \mathbb{S}et$, and models functor Mod : $\text{Sign}^{op} \to \mathbb{C}at$ assigning to any signature $\Sigma \in |\text{Sign}|$ a category $\text{Mod}(\Sigma)$ whose objects are called Σ-models. As usual, the class of objects of a category C is denoted by $|C|$, and abbreviated to C when clear from the context.

3.1. Simple Implementations

The simplest way to design a specification is by expressing the system requirements in a set of sentences over a suitable signature, i.e. as a pair $SP = (Sig(SP), Ax(SP))$ where $Sig(SP) \in |\text{Sign}|$ and $Ax(SP) \subseteq |\text{Sen}(Sig(SP))|$. The (loose) semantics of such a *flat* specification SP is the pair $(Sig(SP), Mod(SP))$ where

$$Mod(SP) = \{M \in |Mod(Sig(SP))| : \ M \models Ax(SP)\}.$$

In this context, a refinement step is understood as a restriction of an abstract class of models to a more concrete one. Following the terminology of Sannella and Tarlecki, we call a specification which refines another one an *implementation*. Formally, a specification SP' is a *simple implementation* of a specification SP over the same signature, in symbols $SP \rightsquigarrow SP'$, whenever $Mod(SP) \supseteq Mod(SP')$. Transitivity of the inclusion relation ensures the *vertical composition* of simple implementation steps.

Example 3. Two refinement steps are illustrated with simple implementations in the $\mathcal{D}^{\downarrow}$ institution. Consider specification SP_0 from Ex. 1 which expresses a few rudimentary requirements for the behavior of a file compressing service. The action set A defined there provides the signature of SP_0; similarly, its axioms are

the three sentences (0.1) - (0.3) in the example.

First refinement step $SP_0 \rightsquigarrow SP_1$. SP_0 is a very loose specification which would allow to start a computation with an arbitrary action. We will be a bit more precise now and require that at the beginning only an input (of a text or gif file) is allowed, as captured by axiom (1.1) below. Moreover whenever an output action (of any kind) happens, the system must go on with an input (of any kind), as in axiom (1.4). This leads to the specification SP_1 with $Sig(SP_1) = Sig(SP_0) = A$ and the following set of axioms $Ax(SP_1)$:

(1.1) $\langle inTxt + inGif \rangle \mathbf{tt} \wedge [outZip + outJpg]\mathbf{ff}$
(1.2) $[A^*; inTxt]\big(\langle outZip \rangle \mathbf{tt} \wedge [-outZip]\mathbf{ff}\big)$
(1.3) $[A^*; inGif]\big(\langle outJpg \rangle \mathbf{tt} \wedge [-outJpg]\mathbf{ff}\big)$
(1.4) $[A^*; (outZip + outJpg)]\big(\langle inTxt + inGif \rangle \mathbf{tt} \wedge [outZip + outJpg]\mathbf{ff}\big)$

It is easy to check that $SP_0 \rightsquigarrow SP_1$ holds: Axioms (0.1) and (0.2) of SP_0 occur as axioms (1.2) and (1.3) in SP_1. It is also easy to see that non-termination (axiom (0.3) of SP_0) is guaranteed by the axioms of SP_1.

The level of underspecification is, at this moment, still very high. Among the multiple models of SP_1, the LTS shown in Fig. 2, with initial state w_0, exhibits an alternating compression mode.

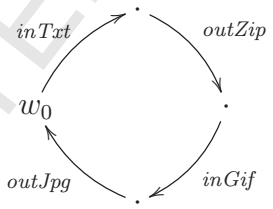


Figure 2: A model of SP_1.

Second refinement step $SP_1 \rightsquigarrow SP_2$. This step rules out alternating behaviours as the one above. The first axiom (2.1) of specification SP_2 is equivalent to axiom (1.1) of SP_1. Alternating behaviours are ruled out by axioms (2.2) and (2.3) which require that, after any text or image compression, the initial state must be reached again. To express this we need state variables and binders which are available in $\mathcal{D}^{\downarrow}$-logic. In our example we introduce one state variable x_0 which names the initial state by using the binder at the beginning of axioms (2.2) and (2.3). Moreover, we only want to admit determinisitic models such that in any (reachable) state there can be no two outgoing transitions labelled with the same action. It turns

out that $\mathcal{D}^{\downarrow}$ makes possible to specify this property with the set of axioms (det) shown below. This leads to the specification SP_2 with $Sig(SP_2) = Sig(SP_1) = A$ and with axioms $Ax(SP_2)$:

(2.1) $(\langle inTxt\rangle \mathbf{tt} \vee \langle inGif\rangle \mathbf{tt}) \wedge [outZip + outJpg]\mathbf{ff}$
(2.2) $\downarrow x_0.\, [inTxt]\big(\langle outZip\rangle x_0 \wedge [-outZip]\mathbf{ff}\big)$
(2.3) $\downarrow x_0.\, [inGif]\big(\langle outJpg\rangle x_0 \wedge [-outJpg]\mathbf{ff}\big)$
(det) For each $a \in A$, the axiom: $[A^*] \downarrow x.(\langle a\rangle \mathbf{tt} \Rightarrow (\langle a\rangle \downarrow y.\, @_x[a]y))$

Clearly, SP_2, shown in Fig. 3, fulfills the requirements of SP_1, i.e. $SP_1 \rightsquigarrow SP_2$. SP_2 has three models which are shown in . (Remember that models can only have states reachable from the initial one.) The first model allows only text compression, the second one does the same for image compression, and the third supports both. The signature of all models is A, though in the first two some actions have no transitions.

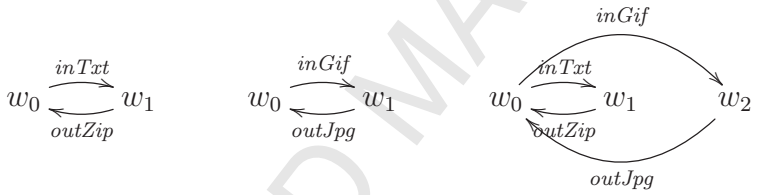


Figure 3: Models of SP_2.

Other variants of SP_2 could be considered to underpin the expressive power of $\mathcal{D}^{\downarrow}$. If we want only the model where both text and image compression are possible, then we can simply replace in axiom (2.1) $\langle inTxt\rangle \mathbf{tt} \vee \langle inGif\rangle \mathbf{tt}$ by $\langle inTxt\rangle \mathbf{tt} \wedge \langle inGif\rangle \mathbf{tt}$. If we would like to require that text compression must be possible in any model but image compression is optional, thus ruling out the second model in Fig. 3, then we would simply omit $\vee\langle inGif\rangle \mathbf{tt}$ in axiom (2.1). This is an interesting case since it shows that $\mathcal{D}^{\downarrow}$ can express the so-called "may"-transitions present in modal transition systems [23] to specify options for implementations.

3.2. Constructor Implementations

The concept of a simple implementation is, in general, too strict to capture software development practice, along which, implementation decisions typically introduce new design features, or reuse already implemented ones, usually entailing a change of signatures along the way. The notion of *constructor implementation* offers the necessary generalization. The idea is that for implementing a specification *SP* one may use a given specification *SP'* and apply a construction to the

models of *SP'* such that they become models of *SP*. More generally, an implementation of *SP* may be obtained by using not only one but several specifications $SP'_1, \ldots, SP'_n$ as a basis and applying an n-ary constructor such that for any tuple of models of $SP'_1, \ldots, SP'_n$ the construction leads to a model of *SP*. Such an implementation is called a *constructor implementation with decomposition* in [32] since the implementation of *SP* is designed by using several components. These ideas are formalized as follows, partially in a less general manner than the correponding definitions in [32] which allow also partial and higher-order functions as constructors.

Given signatures $\Sigma_1, ..., \Sigma_n, \Sigma \in |\mathrm{Sign}|$, a *constructor* is a total function $\kappa : \mathrm{Mod}(\Sigma_1) \times \cdots \times \mathrm{Mod}(\Sigma_n) \to \mathrm{Mod}(\Sigma)$. Constructors compose as follows: Given a constructor $\kappa : \mathrm{Mod}(\Sigma_1) \times \cdots \times \mathrm{Mod}(\Sigma_n) \to \mathrm{Mod}(\Sigma)$ and a set of constructors $\kappa_i : \mathrm{Mod}(\Sigma_i^1) \times \cdots \times \mathrm{Mod}(\Sigma_i^{k_i}) \to \mathrm{Mod}(\Sigma_i)$, $1 \leq i \leq n$, the constructor $\kappa(\kappa_1, \ldots, \kappa_n) : \mathrm{Mod}(\Sigma_1^1) \times \cdots \times \mathrm{Mod}(\Sigma_1^{k_1}) \times \cdots \times \mathrm{Mod}(\Sigma_n^1) \times \cdots \times \mathrm{Mod}(\Sigma_n^{k_n}) \to \mathrm{Mod}(\Sigma)$ is obtained by the usual composition of functions.

Definition 4 (Constructor implementation). Given specifications $SP, SP'_1, \ldots, SP'_n$, and a constructor

$$\kappa : \mathrm{Mod}(Sig(SP'_1)) \times \cdots \times \mathrm{Mod}(Sig(SP'_n)) \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(Sig(SP)),$$

$\langle SP'_1, \ldots, SP'_n \rangle$ is a *constructor implementation via* κ of SP, in symbols $SP \rightsquigarrow_{\kappa} \langle SP'_1, \ldots, SP'_n \rangle$, if for all $M_i \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP'_i)$, $\kappa(M_1, \ldots, M_n) \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP)$. We say that the implementation involves a decomposition if $n > 1$.

3.3. Abstractor Implementations

Often in formal program development properties of a specification are not literally satisfied by an implementation, but only up to an admissible abstraction. Usually such an abstraction concerns implementation details which are hidden from the user of the system and which may, for instance for efficiency reasons, not be fully conform to the requirements specification. In such cases the implementation is still considered to be correct if it shows the desired observable behavior. In general this can be expressed by considering an equivalence relation $\equiv$ on the models of the abstract specification, and requiring the implementation models to be only equivalent to models of the requirements specification.

Formally, let SP be a specification and $\equiv \subseteq \mathrm{Mod}(Sig(SP)) \times \mathrm{Mod}(Sig(SP))$ an equivalence relation. Let $Abs_{\equiv}(\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP))$ be the closure of $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP)$ under $\equiv$. A specification SP', with the same signature as SP is a *simple abstractor implementation* of SP w.r.t. $\equiv$ whenever $Abs_{\equiv}(\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP)) \supseteq \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP')$. Both concepts, constructors and abstractors can be combined as shown in the definition of an abstractor implementation. (For simplicity, the term constructor is omitted.)

Definition 5 (Abstractor implementation). Let $SP, SP'_1, \ldots, SP'_n$ be specifications, $\kappa : \mathrm{Mod}(Sig(SP'_1)) \times \cdots \times \mathrm{Mod}(Sig(SP'_n)) \to \mathrm{Mod}(Sig(SP))$ a constructor, and $\equiv \subseteq \mathrm{Mod}(Sig(SP)) \times \mathrm{Mod}(Sig(SP))$ an equivalence relation. We say that $\langle SP'_1, \cdots, SP'_n \rangle$ is an *abstractor implementation of SP via κ w.r.t. $\equiv$*, in symbols $SP \rightsquigarrow^{\equiv}_{\kappa} \langle SP'_1, \cdots, SP'_n \rangle$, if for all $M_i \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP'_i)$, $\kappa(M_1, \ldots, M_n) \in Abs_{\equiv}(\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(SP))$.

4. Reactive Systems Development with $\mathcal{D}^{\downarrow}$

4.1. Constructor Implementations in $\mathcal{D}^{\downarrow}$

This section introduces a palette of constructors to support the formal development of reactive systems within $\mathcal{D}^{\downarrow}$, instantiating the definitions given in Sect. 3.2. The idea is to lift standard constructions on labelled transition systems (see, e.g. [34]) to constructors for implementations. The constructors introduced in the sequel will be illustrated with our running example.

Along the refinement process it is sometimes convenient to reduce the action set, for instance, by omitting some actions previously introduced as auxiliary actions or as options that are no longer needed. For this purpose we use the *alphabet extension constructor*. Remember that constructors always map concrete models to abstract ones. Therefore when omitting actions in a refinement step we need an alphabet extension on the concrete models to fit them to the abtsract signature.

Definition 6 (Alphabet extension). Let $A, A' \in |\mathrm{Sign}^{\mathcal{D}^{\downarrow}}|$ be signatures in $\mathcal{D}^{\downarrow}$, i.e. action sets, such that $A \subseteq A'$. The *alphabet extension constructor* $\kappa_{ext} : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A) \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A')$ is defined as follows: For each $\mathcal{M} = (W, w_0, R) \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$, $\kappa_{ext}(\mathcal{M}) = (W, w_0, R')$ with $R'_a = R_a$ for all $a \in A$ and $R'_a = \emptyset$ for all $a \in A' \setminus A$.

Example 4. The specification SP_2 of Ex. 3 has the three models shown in Fig. 3. Hence, it allows three directions to proceed further in the product line.

Third refinement step $SP_2 \rightsquigarrow_{\kappa_{ext}} SP_3$. We will consider here the simple case of a service for text compression only. The following specification SP_3 is a direct axiomatisation of the first model in Fig. 3 considered over the smaller action set $A_3 = \{inTxt, outZip\}$. Hence, $Sig(SP_3) = A_3$ and the axioms in $Ax(SP_3)$ are:

(3.1) $\downarrow x_0.\, (\langle inTxt \rangle \downarrow x_1.\, (\langle outZip \rangle x_0 \wedge [inTxt]\mathbf{ff}) \wedge [outZip]\mathbf{ff})$
(det) For each $a \in A_3$, the axiom: $[A_3^*] \downarrow x.(\langle a \rangle \mathbf{tt} \Rightarrow (\langle a \rangle \downarrow y.\, @_x[a]y))$

Since the signature of SP_3 has less actions than the one of SP_2, we apply an alphabet extension constructor $\kappa_{ext} : \mathrm{Mod}^{\mathcal{D}^\downarrow}(A_3) \to \mathrm{Mod}^{\mathcal{D}^\downarrow}(A)$ which transforms the model of SP_3 into an LTS with the same states and transitions but with an empty accessibility relation for the actions in $A \backslash A_3$. Then, trivially, $SP_2 \rightsquigarrow_{\kappa_{ext}} SP_3$ holds. Specification SP_3 is a simple example that shows how labelled transition systems can be directly specified in $\mathcal{D}^\downarrow$. This could suggest that we are already close to a concrete implementation. But this is not true, since SP_3 is in principle just an interface specification which specifies the system behavior "from the outside", i.e. its interactions with the user. □

The standard way to build reactive systems is by aggregating in parallel smaller components. The following parallel composition constructor, synchronising on shared actions, caters for this.

Definition 7 (Parallel composition). Given signatures A and A' the *parallel composition constructor* $\kappa_\otimes : \mathrm{Mod}^{\mathcal{D}^\downarrow}(A) \times \mathrm{Mod}^{\mathcal{D}^\downarrow}(A') \to \mathrm{Mod}^{\mathcal{D}^\downarrow}(A \cup A')$ is a function mapping models $\mathcal{M} = (W, w_0, R) \in \mathrm{Mod}^{\mathcal{D}^\downarrow}(A)$ and $\mathcal{M}' = (W', w_0', R') \in \mathrm{Mod}^{\mathcal{D}^\downarrow}(A')$, to the $A \cup A'$-model $\mathcal{M} \otimes \mathcal{M}' = \big(W^\otimes, (w_0, w_0'), R^\otimes\big)$ where $W^\otimes \subseteq W \times W'$ and $R^\otimes = (R_a^\otimes)_{a \in A \cup A'}$ are the least sets satisfying $(w_0, w_0') \in W^\otimes$, and, for each $(w, w') \in W^\otimes$,

- if $a \in A \cap A'$, $(w, v) \in R_a$, $(w', v') \in R'_a$, then $(v, v') \in W^\otimes$ and $\big((w, w'), (v, v')\big) \in R_a^\otimes$;
- if $a \in A \setminus A'$, $(w, v) \in R_a$, then $(v, w') \in W^\otimes$ and $\big((w, w'), (v, w')\big) \in R_a^\otimes$;
- if $a \in A' \setminus A$, $(w', v') \in R'_a$, then $(w, v') \in W^\otimes$ and $\big((w, w'), (w, v')\big) \in R_a^\otimes$.

Since, up to isomorphism, parallel composition is associative, the extension of this constructor to the n-ary case is straightforward. Parallel composition is a crucial operator for constructor implementations with decomposition; see Def. 4. Remember again that constructors always go from concrete models to abstract ones, i.e. in the opposite direction of the refinement process. Therefore the parallel composition constructor justifies the implementation of reactive systems by decomposition.

Example 5. Let us construct an implementation for the interface specification SP_3 in Ex. 4, based on a decomposition into two components, a controller component *Ctrl* and a component *GZip* which does the actual text compression. The controller has actions $A_{Ctrl} = \{inTxt, txt, zip, outZip\}$. First, it receives a txt-file from the user (action *inTxt*). Then it hands over the text, with action *txt*, to the

GZip component and receives the resulting zip-file (action *zip*). Finally, it returns the zip-file (action *outZip*) and becomes ready to process another compression. Hence, the controller component has signature $Sig(Ctrl) = A_{Ctrl}$. The axioms below specify a single model, shown in Fig. 4 (left), with the intended behavior.

$$\begin{aligned}(4.1)\ \downarrow x_0.\,(\langle inTxt\rangle \downarrow x_1.\,(\langle txt\rangle \downarrow x_2.\,(\langle zip\rangle \downarrow x_3.\,(\langle outZip\rangle x_0 \wedge [-outZip]\mathbf{ff})\\ \wedge[-zip]\mathbf{ff})\\ \wedge[-txt]\mathbf{ff})\\ \wedge[-inTxt]\mathbf{ff})\end{aligned}$$

(det) For each $a \in A_{Ctrl}$, the axiom: $[A^*_{Ctrl}] \downarrow x.(\langle a\rangle \mathbf{tt} \Rightarrow (\langle a\rangle \downarrow y.\, @_x[a]y))$

The *GZip* component has the actions $A_{Gzip} = \{txt, compTxt, zip\}$. First, it receives (action *txt*) the text to be compressed from the controller. Then it does the compression (action *compTxt*), delivers the zip-file (action *zip*) to the controller and is ready for a next round. The *GZip* component has the signature $Sig(Gzip) = A_{Gzip}$ and the axioms *Ax(Gzip)* are similar to the ones of the controller and not shown here. They specify a single model, shown in Fig. 4 (right).

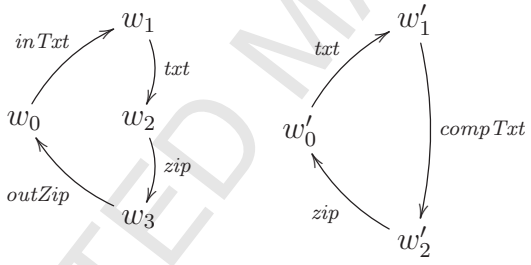


Figure 4: Models of *Ctrl* and *GZip*.

To construct an implementation $\langle Ctrl, GZip\rangle$ by decomposition (see Def. 4), we use the synchronous parallel composition operator "$\otimes$" defined above. According to [32], Exercise 6.1.15, any constructor gives rise to a specification building operation. This means that we can define the specification $Ctrl \otimes GZip$ whose model class consists of all possible parallel compositions of the models of the single specifications. Since *Ctrl* and *GZip* have, up to isomorphism, only one model there is also only one model of $Ctrl \otimes GZip$ which is shown in Fig. 5. Therefore, we know by construction that $Ctrl \otimes GZip \rightsquigarrow_{\kappa_\otimes} \langle Ctrl, GZip\rangle$ is a constructor implementation with decomposition. It remains to fill the gap between SP_3 and $Ctrl \otimes GZip$ which will be done with the action refinement constructor to be introduced in Def. 9.

Two constructions which are frequently used, and typically present in most process algebras, are *relabelling* and *restriction*. They are particular cases of the reduct functor in the $\mathcal{D}^{\downarrow}$ institution.

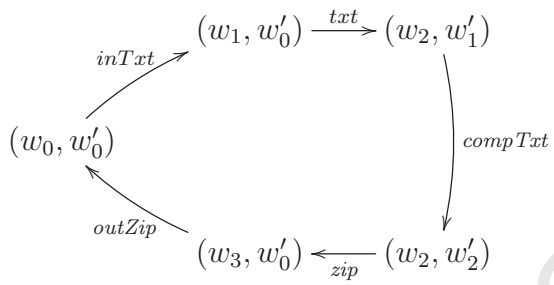


Figure 5: Model of *Ctrl* ⊗ *GZip*.

Definition 8 (Reduct, relabelling and restriction). Let $\sigma : A \to A'$ be a signature morphism. The *reduct constructor* $\kappa_\sigma : \mathrm{Mod}^{\mathcal{D}^\downarrow}(A') \to \mathrm{Mod}^{\mathcal{D}^\downarrow}(A)$ maps any model $\mathcal{M}' \in \mathrm{Mod}^{\mathcal{D}^\downarrow}(A')$ to its reduct $\kappa_\sigma(\mathcal{M}') = \mathrm{Mod}^{\mathcal{D}^\downarrow}(\sigma)(\mathcal{M}')$. Whenever σ is a bijective function, κ_σ is a *relabelling constructor*. If σ is injective, κ_σ is a *restriction constructor* removing actions and transtions.

An important refinement concept for reactive systems is *action refinement* where an abstract action is implemented by a combination of several concrete ones (see [14]). It turns out that an action refinement constructor can be easily defined in $\mathcal{D}^\downarrow$-logic if we use the reduct functor for models over a signature consisting of structured actions built over atomic ones.

Definition 9 (Action refinement). Let $A, A' \in |\mathrm{Sign}^{\mathcal{D}^\downarrow}|$ be signatures in $\mathcal{D}^\downarrow$, i.e. sets of actions. Let D be a finite subset of $\mathrm{Act}(A')$ considered as a signature in $|\mathrm{Sign}^{\mathcal{D}^\downarrow}|$ and let $f : A \to D$ be a signature morphism. The *action refinement constructor* $|_f : \mathrm{Mod}^{\mathcal{D}^\downarrow}(D) \to \mathrm{Mod}^{\mathcal{D}^\downarrow}(A)$ maps any model $\mathcal{M}' \in \mathrm{Mod}^{\mathcal{D}^\downarrow}(D)$ to its reduct $\mathrm{Mod}^{\mathcal{D}^\downarrow}(f)(\mathcal{M}')$.

Example 6. Let us establish a refinement relation between SP_3 (Ex. 4) and $Ctrl \otimes GZip$ (Ex. 5). The signature of SP_3 consists of actions $A_3 = \{inTxt, outZip\}$, the signature of $Ctrl \otimes GZip$ is the set $A_4 = \{inTxt, txt, compTxt, zip, outZip\}$. To obtain an action refinement, we define the signature morphism $f : A_3 \to \mathrm{Act}(A_4)$ by $f(inTxt) = inTxt; txt; compTxt$ and $f(outZip) = zip; outZip$. Then, we apply the action refinement constructor $|_f : \mathrm{Mod}^{\mathcal{D}^\downarrow}(A_4) \to \mathrm{Mod}^{\mathcal{D}^\downarrow}(A_3)$ induced by f. Clearly, the application of $|_f$ to the model of $Ctrl \otimes GZip$ leads to the model of SP_3 explained above. Hence, $SP_3 \rightsquigarrow_{|_f} Ctrl \otimes GZip$, which combined with Ex. 5, justifies $Ctrl \otimes GZip \rightsquigarrow_{\kappa_\otimes} \langle Ctrl, GZip \rangle$ which completes a refinement chain:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow SP_2 \rightsquigarrow_{\kappa_{ext}} SP_3 \rightsquigarrow_{|_f} Ctrl \otimes GZip \rightsquigarrow_{\kappa_\otimes} \langle Ctrl, GZip \rangle.$$

Finally, let us discuss how the last specification in the chain could be implemented in a concrete process algebra. Translation from $\mathcal{D}^{\downarrow}$ to FSP yields

```
Ctrl = (inTxt -> txt -> zip -> outZip -> Ctrl).
Gzip = (txt -> compTxt -> zip -> Gzip).
```

The FSP semantics of the two processes are just the two models of the *Ctrl* and *Gzip* specifications respectively. They can be put together to form a concurrent system `(Ctrl || Gzip)` by using the synchronous parallel composition of FSP processes. Since the semantics of parallel composition in FSP coincides with the one of constructor $\kappa_{\otimes}$, we conclude that the FSP system `(Ctrl || Gzip)` is a correct implementation of the interface specification SP_3.

4.2. Abstractor Implementations in $\mathcal{D}^{\downarrow}$

Abstractor implementations in the field of algebraic specifications use typically observational equivalence relations between algebras based on the evaluation of terms with observable sorts. Interestingly, in the area of concurrent systems, abstractors have a very intuitive interpretation in terms of *bisimilarity* (aka bisimulation equivalence). Let us briefly recall this standard notion [27]:

Definition 10 (Bisimilarity). Given two models $\mathcal{M} = (W, w_0, R)$ and $\mathcal{M}' = (W', w_0', R')$ for signature A, a *bisimulation* between $\mathcal{M}$ and $\mathcal{M}'$ is a relation $B \subseteq W \times W'$ that contains (w_0, w_0'), and is such that

(zig) for any $a \in A$, $w, v \in W$, $w' \in W'$, such that $(w, w') \in B$, if $(w, v) \in R_a$, then there is a $v' \in W'$ such that $(w', v') \in R'_a$ and $(v, v') \in B$;

(zag) for any $a \in A$, $w \in W$, $w', v' \in W'$, such that $(w, w') \in B$, if $(w', v') \in R'_a$, then there is a $v \in W$ such that $(w, v) \in R_a$ and $(v, v') \in B$.

The *bisimilarity* relation with respect to A, is the equivalence $\equiv_A \subseteq \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A) \times \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$ defined as

$$\equiv_A \triangleq \{(\mathcal{M}_1, \mathcal{M}_2) \mid \text{there is a bisimulation between } \mathcal{M}_1 \text{ and } \mathcal{M}_2\}.$$

subscript A is omitted when the context is clear.

There is a number of well known properties of bisimulations that are used in the sequel. In particular, bisimulations are closed for composition, converse and union, and form a complete lattice whose top coincides with bisimilarity.

To motivate the use of an abstractor implementation for bisimilarity, let us consider the specification $SP = (\{a\}, \{\downarrow x.\langle a \rangle x\})$. The axiom is satisfied by the

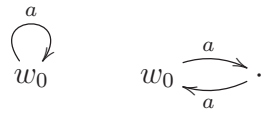


Figure 6: Behaviourally equivalent LTSs.

first model in Figure 6, but not by the second one. Clearly, however, both are bisimilar and so it should be irrelevant, for implementation purposes, to choose one or the other as an implementation of SP.

Vertical composition of implementations refers to the situation where the implementation of a specification is further refined in a subsequent step. For simple implementations it is trivial to show that two implementation steps compose. In the context of constructor and abstractor implementations the situtation is more complex. A general condition to obtain vertical composition in this case was established in [31]. However, the original result was only given for unary implementation constructors. In order to adopt parallel composition as a constructor, we first generalise the institution independent result of [31] to the n-ary case involving decomposition:

Theorem 3 (Vertical composition). Consider specifications $SP, SP_1, \ldots, SP_n$ over an arbitrary institution, a constructor

$$\kappa : \mathrm{Mod}(Sig(SP_1)) \times \cdots \times \mathrm{Mod}(Sig(SP_n)) \to \mathrm{Mod}(Sig(SP)),$$

and an equivalence $\equiv \subseteq \mathrm{Mod}(Sig(SP)) \times \mathrm{Mod}(Sig(SP))$ such that $SP \rightsquigarrow_{\kappa}^{\equiv} \langle SP_1, \cdots, SP_n \rangle$. For each $i \in \{1, \ldots, n\}$, let $SP_i \rightsquigarrow_{\kappa_i}^{\equiv_i} \langle SP_i^1, \cdots, SP_i^{k_i} \rangle$ with specifications $SP_i^1, \ldots, SP_i^{k_i}$, constructor

$$\kappa_i : \mathrm{Mod}(Sig(SP_i^1)) \times \cdots \times \mathrm{Mod}(Sig(SP_i^{k_i})) \to \mathrm{Mod}(Sig(SP_i)),$$

and equivalence $\equiv_i \subseteq \mathrm{Mod}(Sig(SP_i)) \times \mathrm{Mod}(Sig(SP_i))$. Suppose that κ preserves the abstractions $\equiv_i$, i.e. for each $\mathcal{M}_i, \mathcal{N}_i \in Mod(Sig(SP_i))$ such that $\mathcal{M}_i \equiv_i \mathcal{N}_i$, $\kappa(\mathcal{M}_1, \ldots, \mathcal{M}_n) \equiv \kappa(\mathcal{N}_1, \ldots, \mathcal{N}_n)$. Then,

$$SP \rightsquigarrow_{\kappa(\kappa_1, \cdots, \kappa_n)}^{\equiv} \langle\, SP_1^1, \cdots, SP_1^{k_1}, \cdots, SP_n^1, \cdots, SP_n^{k_n} \rangle.$$

Proof. For each $1 \leq i \leq n$ and for all $1 \leq j \leq k_i$, let $\mathcal{M}_i^j \in Mod(SP_i^j)$. By hypothesis, for each i, $\mathcal{M}_i \equiv_i \kappa_i(\mathcal{M}_i^1, \ldots, \mathcal{M}_i^{k_i})$, for some model $\mathcal{M}_i \in Mod(SP_i)$. Since κ preserves abstraction $\equiv_i$, $1 \leq i \leq n$,

$$\kappa(\mathcal{M}_1, \ldots, \mathcal{M}_n) \equiv \kappa(\kappa_1(\mathcal{M}_1^1, \ldots, \mathcal{M}_1^{k_1}), \cdots, \kappa_n(\mathcal{M}_n^1, \ldots, \mathcal{M}_n^{k_n})).$$

Since $\kappa(\mathcal{M}_1, \ldots, \mathcal{M}_n) \in Abs_{\equiv}(Mod(SP))$, we get

$$\kappa(\kappa_1, \ldots, \kappa_n)(\mathcal{M}_1^1, \ldots, \mathcal{M}_1^{k_1}, \cdots, \mathcal{M}_n^1, \ldots, \mathcal{M}_n^{k_n}) \in Abs_{\equiv}(Mod(SP)).$$

The remaining results establish the necessary compatibility between the constructors defined in $\mathcal{D}^{\downarrow}$ and behavioural equivalence $\equiv_A \subseteq |\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)| \times |\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)|$, for $A \in \mathrm{Sign}^{\mathcal{D}^{\downarrow}}$, defined as bisimilarity.

Theorem 4. The alphabet extension constructor κ_{ext} preserves behavioural equivalences, i.e. for any $\mathcal{M}_1 \equiv_A \mathcal{M}_2$, $\kappa_{ext}(\mathcal{M}_1) \equiv_{A'} \kappa_{ext}(\mathcal{M}_2)$.

Proof. By hypothesis, since $\mathcal{M}_1 \equiv_A \mathcal{M}_2$, there is at least a bisimulation $B \subseteq W_1 \times W_2$. Then, B is also a bisimulation between $\kappa_{ext}(\mathcal{M}_1)$ and $\kappa_{ext}(\mathcal{M}_2)$. Thus, for all actions $a \in A' \setminus A$, the bisimulation conditions hold trivially. Therefore $\kappa_{ext}(\mathcal{M}_1) \equiv_{A'} \kappa_{ext}(\mathcal{M}_2)$. □

Theorem 5. The parallel composition constructor $\kappa_{\otimes}$ preserves behavioural equivalences, i.e. for any $\mathcal{M}_1 \equiv_{A_1} \mathcal{M}_1'$ and $\mathcal{M}_2 \equiv_{A_2} \mathcal{M}_2'$, $\mathcal{M}_1 \otimes \mathcal{M}_2 \equiv_{A_1 \cup A_2} \mathcal{M}_1' \otimes \mathcal{M}_2'$.

Proof. Suppose, without lost of generality, that $\mathcal{M}_1 \equiv_1 \mathcal{M}_1'$ and $\mathcal{M}_2 \equiv_2 \mathcal{M}_2'$, given the existence of bisimulations B_1 and B_2, respectively. Consider relation $\sim \subseteq (\mathcal{M}_1 \otimes \mathcal{M}_2) \times (\mathcal{M}_1' \otimes \mathcal{M}_2')$ such that $(w_1, w_2) \sim (w_1', w_2')$ if $w_1 \equiv_1 w_1'$ and $w_2 \equiv_2 w_2'$. We prove that $\sim$ is a bisimulation. First note that $(w_{10}, w_{20}) \sim (w_{10}', w_{20}')$ since $(w_{10}, w_{10}') \in B_1$ and $(w_{20}, w_{20}') \in B_2$.

In order to prove the *zig* condition (the proof is similar for the *zag* case) we consider two kinds of admissible transitions:

1. Suppose that $a \in A_1 \cap A_2$, $(w_1, v_1) \in R_a^1$ and $(w_2, v_2) \in R_a^2$. Then a transition $\big((w_1, w_2), (v_1, v_2)\big) R_a^{\otimes}$. By *zig* in B_1, there is a $v_1' \in W_1'$ such that $(v_1, v_1') \in B_1$ and $(w_1', v_1') \in R_a'^1$. Analogously, there is a $v_2' \in W_2'$ such that $(v_2, v_2') \in B_2$ and $(w_2', v_2') \in R_a'^2$. By definitions of $R'^{\otimes}$ and $\sim$, $\big((w_1', w_2'), (v_1', v_2')\big) \in R_a'^{\otimes}$ and $(v_1, v_2) \sim (v_1', v_2')$.
2. Suppose that $a \in A_1 \setminus A_2$ and $(w_1, v_1) \in R_a^1$. By *zig* in B_1, there is a $v_1' \in W_1'$ such that $(v_1, v_1') \in B_1$ and $(w_1', v_1') \in R_a'^1$. Moreover, by definition of relational converse, $R'^{\otimes}$, $\big((w_1', w_2'), (v_1', w_2')\big) \in R_a'^{\otimes}$. Clearly, $w_2 \equiv_2 w_2'$. Therefore, $(v_1, w_2) \sim (v_1', w_2')$. For transitions $a \in A_2 \setminus A_1$, the proof is analogous.

□

Theorem 6. Let $f : A \to \mathrm{Act}(A')$ be a signature morphism. The constructor $|_f$ preserves behavioural equivalences, i.e. for any $\mathcal{M}_1, \mathcal{M}_2 \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\mathrm{Act}(A'))$, if $\mathcal{M}_1 \equiv_{\mathrm{Act}(A')} \mathcal{M}_2$, then $|_f(\mathcal{M}_1) \equiv_A |_f(\mathcal{M}_2)$.

Proof. Assuming $\mathcal{M}_1 \equiv_{A'} \mathcal{M}_2$, let us consider a bisimulation B between M_1 and M_2. We prove that the restriction of B to $W^1 \times W^2$ is a bisimulation between $|_f(\mathcal{M}_1)$ and $|_f(\mathcal{M}_2)$. For the forward direction (cf., the *zig* component), suppose $(w_1, w_1') \in R^1_a$ and $(w_1, w_2) \in B \cap (W_1 \times W_2)$. By $|_f$ definition, $(w_1, w_1') \in R'^1_{f(a)}$ and hence, by the observation below, $(w_2, w_2') \in R'^2_{f(a)}$ for some $(w_1', w_2') \in B$. Since $R'^2_{f(a)} = R^2_a$ and $w_2 \in W_2$, $(w_1', w_2') \in B \cap (W_1 \times W_2)$. The proof for the other direction (*zag*) is analogous. The same argument still applies when considering transitions indexed by sequences of (sets of) actions, as in [1] for string bisimulation. □

5. On the Expressive Power of $\mathcal{D}^{\downarrow}$

In the last section, bisimilarity is taken as a suitable equivalence relation for constructing abstractor implementations. Indeed, this is a usual notion of equivalence for transition systems. In standard modal logic it has a logical counterpart, often referred to as the *Hennessy-Milner property*: bisimilar states satisfy exactly the same modal sentences and, conversely, in two image-finite models (i.e. in which any state has at most finitely many outgoing transitions) any two states satisfying the same modal sentences are bisimilar. Obviously, the latter implication does also hold in $\mathcal{D}^{\downarrow}$-logic since Henessy-Milner logic is a fragment of $\mathcal{D}^{\downarrow}$ (where, anyway, sentences are only interpreted in the initial state). However, the first implication of the Hennessy-Milner property does not hold in $\mathcal{D}^{\downarrow}$: the logic fails to be modally invariant, i.e. bisimilar states do not necessarily satisfy the same $\mathcal{D}^{\downarrow}$-sentences. A counterexample was presented in Sect. 4.2, Fig. 6. The first model satisfies the sentence $\downarrow x.\langle a\rangle x$ but the second one doesn't. This is not a surprise since $\mathcal{D}^{\downarrow}$-logic is a very powerful logic. If we want to abstract from a specification w.r.t. bisimulation equivalence then we can use an abstractor implementation as explained in Sect. 4.2. Indeed the concept of an abstractor implementation would be meaningless if sentences of $\mathcal{D}^{\downarrow}$-logic were preserved by bisimulation equivalence.

In this section we discuss the expressive power of $\mathcal{D}^{\downarrow}$-logic and show that it allows us to specify finite A-models uniquely up to isomorphism. Since the converse direction also holds, i.e. isomorphic models satisfy the same A-sentences, $\mathcal{D}^{\downarrow}$-logic is as powerful as model isomorphism to distinguish finite A-models.

Model morphisms were defined in Sect. 2.2. Two A-models $\mathcal{M}, \mathcal{M}' \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$ are *isomorphic*, in symbols $\mathcal{M}$ **iso** $\mathcal{M}'$, if there is a pair of morphisms $h : \mathcal{M} \to \mathcal{M}'$

and $h^{-1} : \mathcal{M}' \to \mathcal{M}$ such that $h \cdot h^{-1} = id_{\mathcal{M}}$ and $h^{-1} \cdot h = id_{\mathcal{M}'}$. The following result was originally presented in [18]:

Theorem 7. Let $\mathcal{M}$ and $\mathcal{M}'$ be A-models such that $\mathcal{M}$ **iso** $\mathcal{M}'$. Then, for any A-sentence φ, we have

$$\mathcal{M} \models \varphi \text{ iff } \mathcal{M}' \models \varphi.$$

For the remainder of this section we assume given a finite, non-empty set A of actions and two *finite A-models* $\mathcal{M} = (W, w_0, R)$ and $\mathcal{M}' = (W', w_0', R')$, i.e. the sets W, W' are finite. We show that there exists an A-sentence $\varphi_{\mathcal{M}}$ which determines $\mathcal{M}$ up to isomorphism. $\varphi_{\mathcal{M}}$ is constructed as follows:

$$\varphi_{\mathcal{M}} = \downarrow w_0.\, \mathtt{F}(w_0, \mathtt{Im}(w_0), W, \{w_0\})$$

where the initial state w_0 is introduced as a bound variable, $\mathtt{F}$ is algorithmically defined in Table 1, and for any state $w \in W$, $\mathtt{Im}(w) = \{(a, v) \in A \times W \mid (w, v) \in R_a\}$ and, in the following algorithm, for any $a \in A$, $\mathtt{Im}(w, a) = \{v \in W \mid (a, v) \in \mathtt{Im}(w)\}$. The algorithm takes the model $\mathcal{M}$ and performs a recursive breadth-first traversal starting from the initial state of $\mathcal{M}$. For each reached state w it checks its outgoing transitions and requires the existence of such transitions in the formula. Additionally it requires that no other transitions with source state w exist. If all states are visited the algorithm terminates by requiring that the states of $\mathcal{M}$ are pairwise different. The algorithm uses the states of $\mathcal{M}$ as variables. Whenever a new state v is reached, v is bound with the binder of $\mathcal{D}^{\downarrow}$-logic.

Example 7. As an example, let $\mathcal{M}$ be the model on the right in Fig. 3. We show how $\varphi_{\mathcal{M}}$ can be derived by using the algorithm in Table 1.

$$\varphi_{\mathcal{M}} = \downarrow w_0.\, \mathtt{F}(w_0, \mathtt{Im}(w_0), W, \{w_0\})$$

with $\mathtt{Im}(w_0) = \{(inTxt, w_1), (inGif, w_2)\}$ and $W = \{w_0, w_1, w_2\}$. Then we compute:

$$\begin{aligned}
&\mathtt{F}(w_0, \mathtt{Im}(w_0), W, \{w_0\}) = \\
&@_{w_0}\langle inTxt\rangle \downarrow w_1.\, \mathtt{F}(w_0, \{(inGif, w_2)\}, W, \{w_0, w_1\}) = \\
&@_{w_0}\langle inTxt\rangle \downarrow w_1.@_{w_0}\langle inGif\rangle \downarrow w_2.\, \mathtt{F}(w_0, \emptyset, W, W) = \\
&@_{w_0}\langle inTxt\rangle \downarrow w_1.@_{w_0}\langle inGif\rangle \downarrow w_2. \\
&\qquad @_{w_0}([inTxt]w_1 \wedge [inGif]w_2 \wedge [outZip]\mathbf{ff} \wedge [outJpg]\mathbf{ff}) \wedge \\
&\qquad \mathtt{F}(w_1, \mathtt{Im}(w_1), \{w_1, w_2\}, W)
\end{aligned}$$

F(w, ImageToVisit, StatesToVisit, BoundStates) =
  if ImageToVisit $\neq \emptyset$
  then {
    //take a transition outgoing from w and specify that it is required;
    //if the target state v has been introduced as a bound variable before
    //then require $@_w\langle a\rangle v$ and continue;
    //otherwise bind v as a variable, require $@_w\langle a\rangle \downarrow v$ and continue;
    choose $(a, v) \in$ ImageToVisit;
    if $v \in$ BoundStates
    then return $@_w\langle a\rangle v \wedge$
      F(w, ImageToVisit $\setminus\{(a, v)\}$, StatesToVisit, BoundStates)
    else return $@_w\langle a\rangle \downarrow v.$
      F(w, ImageToVisit $\setminus\{(a, v)\}$, StatesToVisit, BoundStates $\cup\{v\}$)
  }
  else {
    //i.e. ImageToVisit $= \emptyset$, which means that all transitions outgoing
    //from w are already specified;
    //then finalise the visit of w by requiring that only the transitions
    //outgoing from w are allowed at w and continue with some other
    //state v which has been bound before but not yet visited
    //if such a state exists;
    //otherwise terminate by specifying that all states in W are different;
    let finalise(w) $= @_w(\bigwedge_{a\in A}[a](\bigvee_{u\in Im(w,a)} u))$;
    StatesToVisit = StatesToVisit $\setminus \{w\}$;
    if StatesToVisit $\neq \emptyset$
    then {
      choose $v \in$ BoundStates $\cap$ StatesToVisit;
      return finalise(w) $\wedge$ F(v, Im(v), StatesToVisit, BoundStates)
    }
    else return finalise(w) $\wedge \bigwedge_{w\neq w'\in W} \neg @_w w'$
  }

where $\bigvee_{u\in\emptyset}$ stands for **ff**.

Table 1: Algorithm to construct an A-sentence.

where

$$\begin{aligned}
&\mathtt{F}(w_1, \mathtt{Im}(w_1), \{w_1, w_2\}, W) = \\
&@_{w_1}\langle outZip\rangle w_0 \wedge \mathtt{F}(w_1, \emptyset, \{w_1, w_2\}, W) = \\
&@_{w_1}\langle outZip\rangle w_0 \wedge \\
&\quad @_{w_1}([inTxt]\mathbf{ff} \wedge [inGif]\mathbf{ff} \wedge [outZip]w_0 \wedge [outJpg]\mathbf{ff}) \wedge \\
&\quad \mathtt{F}(w_2, \mathtt{Im}(w_2), \{w_2\}, W)
\end{aligned}$$

where

$$\begin{aligned}
&\mathtt{F}(w_2, \mathtt{Im}(w_2), \{w_2\}, W) = \\
&@_{w_2}\langle outJpg\rangle w_0 \wedge \mathtt{F}(w_2, \emptyset, \{w_2\}, W) = \\
&@_{w_2}\langle outJpg\rangle w_0 \wedge \\
&\quad @_{w_2}([inTxt]\mathbf{ff} \wedge [inGif]\mathbf{ff} \wedge [outZip]\mathbf{ff} \wedge [outJpg]w_0) \wedge \\
&\quad \neg@_{w_0}w_1 \wedge \neg@_{w_0}w_2 \wedge \neg@_{w_1}w_2
\end{aligned}$$

Theorem 8. Let $\mathcal{M}$ and $\mathcal{M}'$ be two finite A-models such that $\mathcal{M} \models \varphi$ iff $\mathcal{M}' \models \varphi$ for all A-sentences φ. Then $\mathcal{M}$ **iso** $\mathcal{M}'$.

Proof. We give only a sketch of the proof. Let $\varphi_{\mathcal{M}}$ be the A-sentence derived from $\mathcal{M}$, as explained above. $\mathcal{M}$ and $\mathcal{M}'$ satisfy the same A-sentences and therefore $\mathcal{M}'$ satisfies, in particular, $\varphi_{\mathcal{M}}$. Since $\varphi_{\mathcal{M}}$ specifies $\mathcal{M}$ uniquely up to isomorphism, we get $\mathcal{M}$ **iso** $\mathcal{M}'$. □

6. A Proof System for $\mathcal{D}^{\downarrow}$

This section introduces a proof system for $\mathcal{D}^{\downarrow}$, which, as explained before, combines a hybrid logic with binders $H(@, \downarrow)$ [6], but no propositional symbols (i.e. neither propositions nor nominals), with dynamic logic [12]. The proof system reflects this combination by putting together the proof systems of both components. First, because all state symbols considered are variables, the axioms of $H(@, \downarrow)$ are restricted to state variables, instead of state variables and nominals, as one would expect. On the other hand, the dynamic part consists just of four axioms, expressing how composite programs behave. Axioms involving tests are omitted, as tests themselves are not allowed in the logic. The non-dynamic part of the axiomatics disregards the way programs are built. It introduces a modality symbol for each program in $Act(A)$, and not just for the atomic ones. Thus, the logic can be taken as a multimodal logic with an infinite set of modality symbols $Act(A)$. The proof system is as follows,

Axioms

Basic Kripke axioms:

(Taut) all propositional tautologies
(K) $[\alpha](\varphi \to \psi) \to ([\alpha]\varphi \to [\alpha]\psi)$

These are the axioms of a normal multimodal logic. The next three sets of axioms come from the axiomatization of hybrid logic (cf. [6]).

Axioms for @:

($\mathrm{K}_@$) $@_s(\varphi \to \psi) \to (@_s\varphi \to @_s\psi)$
($@_s$-self-dual) $@_s\varphi \leftrightarrow \neg@_s\neg\varphi$
(Introduction) $(s \wedge \varphi) \to @_s\varphi$

Axioms for the modal theory of labeling:

(Label) $@_s s$
(Nom) $@_s t \to (@_t\varphi \to @_s\varphi)$
(Swap) $@_s t \leftrightarrow @_t s$
(Scope) $@_t@_s\varphi \leftrightarrow @_s\varphi$

Axioms for the interaction between @ and $\diamond$:

(Back) $\langle\alpha\rangle@_s\varphi \to @_s\varphi$
(Bridge) $(\langle\alpha\rangle s \wedge @_s\varphi) \to \langle\alpha\rangle\varphi$

The axioms expressing how binders behave are taken from [6]:

Axioms for binders:

(b1) $\downarrow x.(\varphi \to \psi) \to (\varphi \to \downarrow x.\psi)$
(b2) $\downarrow x.\varphi \to (s \to \varphi[s/x])$
(b3) $\downarrow x.(x \to \varphi) \to \downarrow x.\varphi$
(b4) $\downarrow x.\varphi \leftrightarrow \neg\downarrow x.\neg\varphi$ (self-dual)

Finally, the axioms for composition of programs come from dynamic logic.

Axioms of dynamic logic:

(Comp) $[\alpha;\beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$
(Alt) $[\alpha + \beta] \leftrightarrow ([\alpha]\varphi \wedge [\beta]\varphi)$
(Mix) $[\alpha^*]\varphi \to \varphi \wedge [\alpha][\alpha^*]\varphi$
(Ind) $[\alpha^*](\varphi \to [\alpha]\varphi) \to (\varphi \to [\alpha^*]\varphi)$

where x, s, t are variables, φ and ψ are arbitrary formulas and $\alpha, \beta \in Act(A)$. Note that in (b1) φ cannot contain free occurrences of x. Similarly, in (b2) s must be substituible for x in φ.
The rules are as expected:

Rules:
Modus ponens: $\dfrac{\varphi \to \psi, \varphi}{\psi}$
Necessitation: $\dfrac{\varphi}{[\alpha]\varphi}$
Variable localization: $\dfrac{\varphi}{\downarrow x.\varphi}$
$@_s$-necessitation: $\dfrac{\varphi}{@_s\varphi}$
Paste rules: $\dfrac{@_s(t \wedge \varphi) \to \theta}{@_s\varphi \to \theta}$ $\quad$ $\dfrac{@_s\langle\alpha\rangle(t \wedge \varphi) \to \theta}{@_s\langle\alpha\rangle\varphi \to \theta}$

where t is a variable different from s, that does not occur in either φ or θ.

The first two rules are the rules of a normal multimodal logic, the last two come from hybrid logic. The rule of variable localization is the usual generalization rule for binding.

Deductions are defined in the usual way.

Definition 11. A **deduction** of φ is a finite sequence $\xi_1, ..., \xi_n$ of formulas in $\mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A)$ such that

- for every $1 \leq i \leq n-1$, either ξ_i is an axiom, or ξ_i is obtained from previous expressions in the sequence using a rule, and
- $\xi_n = \varphi$.

We write $\vdash \varphi$, and call φ a *theorem*, whenever such a sequence exists.

The soundness of this proof system is not difficult to prove. As mentioned above, this can be considered as a logic over a multimodal language with a set of modality symbols $Act(A)$. Its models can naturally be regarded as models of such a multimodal language. The interpretation of modalities corresponding to non atomic programs is defined by $R_{\alpha;\alpha'} = R_\alpha \cdot R_{\alpha'}$, $R_{\alpha+\alpha'} = R_\alpha \cup R_{\alpha'}$ and $R_{\alpha^*} = (R_\alpha)^\star$, as defined in Section 2). Given that the proof system for (multimodal) hybrid logic with binders $H(@,\downarrow)$ is sound, we may conclude the validity of the non dynamic axioms and rules. For the dynamic component the result follows as a consequence of the definition of the interpretation of the operators over programs. Thus,

Theorem 9 (Soundness). If $\vdash \varphi$ then $\mathcal{M} \models \varphi$ for any model $\mathcal{M}$.

Discussion on completeness. Establishing completeness seems to be harder. In [6] the authors presented a proof of completeness of a logic similar to $\mathcal{D}^{\downarrow}$, which

they denote by $\mathcal{H}[\downarrow,@](K)$. There are three main differences: first they consider nominals and variables (not only variables as in DH), do not have composition of programs and their models do not have reachability restrictions. This does not allow a straightforward adaptation of their proof to the case at hands.

Let us discuss some problems arising when trying to combine the proof of completeness of hybrid logic with binders with the corresponding proof for dynamic logic. We will revisit the proof for hybrid logic with binders and will point out the problems arising in our setting. The reader not familiar with hybrid logic, can check the details in [6]. The standard proof of completeness of modal logic (and general extensions of modal logic, like hybrid logic) is a consequence of the following fact: *Every consistent set of formulas (in a countable language) is satisfiable in a (countable) model.*

Hence, we have tried to prove a similar result for our logic. More specifically, we have been trying to prove extend/adapt the proof given in reference [6] to the dynamic logic with binders discussed in this paper.

First of all, as in dynamic logic, we have to consider $\mathcal{D}^{\downarrow}$ as a multimodal logic with modality symbols indexed by elements of $Act(A)$, and build the associated canonical model (see below). In general, this is not a dynamic standard one, because R_{α^*} is not equal to $(R_\alpha)^\star$. Recall that a model of a multimodal logic is *dynamic standard* if it is a model of the dynamic logic such that the interpretation of non atomic programs is obtained from the interpretation of the atomic ones by means of the corresponding operations on the associated relations. In the strictly dynamic case, one needs to perform an adequate filtration in order to obtain a dynamic standard model that satisfies the original consistent set.

Typically, in modal logic the states of the canonical model $M^c = (S^c, (R^c_\alpha)_{\alpha \in Act(A)})$ are the *maximal consistent sets* (MCS), i.e. consistent sets which are maximal with respect to inclusion, and the accessibility relation is defined by $sR^c_\alpha t$ iff $\{\varphi \mid [\alpha]\varphi \in s\} \subseteq t$. Note that in the context of this paper there is an additional requirement: the model must be reachable, as well. Here the first problem arises: It is not clear if the canonical model is reachable. Moreover, we should also consider how to deal with initial states.

The completeness proof for hybrid logic (with or without binders) considers only MCS which are *labelled*, in the sense that one of its elements is a state symbol. Then, the proof proceeds by showing that each consistent set can be extended to a MCS labeled by a nominal. This is another problem in our case, since $\mathcal{D}^{\downarrow}$ has variables only.

The extended Lindenbaum's lemma – *Any consistent set of formulas* Γ *can be extended to a maximal consistent set with three desirable properties: labeled by a nominal, pasted (see [6]), and maximal consistent*, plays a crucial role in the classical proof. Then, given a pasted maximal consistent set Γ, labeled by

a nominal, we define *the labeled model yielded by* Γ as $M = (S^\Gamma, (R_a^\Gamma)_{a \in Act(A)})$, where $S^\Gamma = \{\{\varphi \mid @_s\varphi \in \Gamma\} \mid s \text{ is a state symbol}\}$, R_a^Γ is the restriction of R^c to S^Γ, and the natural assignment $g : X \to S^\Gamma$ is given by $g(x) = \{s \in S^\Gamma \mid x \in s\}$. This model is the one that works in standard hybrid logic (multimodal case). Its construction shows that every consistent set of formulas in the multimodal language $Act(A)$ is satisfiable in a model with respect to a assignment function.

To sum up, we have not been able to obtain the completeness proof. There are two main questions to overcome: (a) the canonical models have to be reachable and exhibit an initial state, and (b) $\mathcal{D}^\downarrow$ has no nominals and hence MCS cannot be labelled by them. Concerning the latter, we conjecture that variables can be used, instead of nominals, in the process.

7. Conclusions and Future Work

Building on our previous work [24], this paper completed the characterization of a new logic $\mathcal{D}^\downarrow$ intended to specify abstract requirements for reactive systems, as well as concrete designs expressing (recursive) process structures. Therefore $\mathcal{D}^\downarrow$ is appropriate to instantiate Sannella and Tarlecki's refinement framework to provide stepwise, correct-by-construction development of reactive systems. We have illustrated this with a simple example using specifications and implementation constructors over $\mathcal{D}^\downarrow$. We believe that a case was made for the suitability of both the logic and the method as a viable alternative to other, more standard approaches to the design of reactive software.

Related Work. Since the 80's, the formal development of reactive, concurrent systems has emerged as one of the most active research topics in Computer Science, with a plethora of approaches and formalisms. For a proper comparison with this work, the following paragraphs restrict to two classes of methods: the ones built on top of logics formalised as institutions, and the attempts to apply to the domain of reactive systems the methods and techniques inherited from the loose specification of abstract data types.

In the first class, references [10, 28, 8] introduce different institutions for temporal logics, as a natural setting for the specification of abstract properties of reactive processes. Process algebras themselves have also been framed as institutions. Reference [30] formalises CSP [20] in this way. What distinguishes our own approach, based on $\mathcal{D}^\downarrow$, is the possibility to combine and express in the same logic both abstract properties, as in temporal logics, and their realisation in concrete, recursive process terms, as typical in process algebras.

Our second motivation was to discuss how institution-independent methods, used in (data-oriented) software development, could be applied to the design of reactive systems. A related perspective is proposed in reference [26], which suggests the loose specification of processes on top of the CSP institution [30] mentioned above. The authors explore the reuse of institution independent structuring mechanisms introduced in the CASL framework [3] to develop reactive systems; in particular, process refinement is understood as inclusion of classes of models. Note that the CASL (in-the-large) specification structuring mechanisms can be also taken as specific constructors, as the ones given in this paper.

Future Work. A lot of work, however, remains to be done. For example, decidability of $\mathcal{D}^{\downarrow}$ is yet an open question. In [2] it has been shown that nominal-free dynamic logic with binders is undecidable. But while [2] considers standard Kripke structures and global satisfaction, $\mathcal{D}^{\downarrow}$ takes reachable models and satisfaction with respect to the initial states.

It would also be worthwhile to discuss satisfaction up to some notion of observational equivalence, as done in [5] for algebraic specifications, thus leading to a behavioural version of $\mathcal{D}^{\downarrow}$. Such a behavioural setting offers an interesting way to recover modal invariance for $\mathcal{D}^{\downarrow}$, as recently explored in [18].

The study of initial semantics (for some fragments) of $\mathcal{D}^{\downarrow}$ is also in our research agenda. For example, theories in the fragment of $\mathcal{D}^{\downarrow}$ that alternates binders with diamond modalities (thus binding all visited states) can be shown to have weak initial semantics, which becomes strong initial in a deterministic setting. The abstract study of initial semantics in hybrid(ised) logics reported in [9], together with the canonical model construction for propositional dynamic logic introduced in [22] can offer a nice starting point for this task. Moreover, for handling more complex systems, data must also be represented in the logic.

A second line of inquiry is more directly related to the development method. For example, defining an abstractor on top of some form of weak bisimilarity would allow for a proper treatment of *hiding*, an important operation in CSP [20] and some other process algebras through which a given set of actions is made non observable. Finally, our aim is to add a final step to the method proposed here in which any constructive specification can be translated to a process algebra expression, as currently done by our proof-of-concept translator D2FSP. A particularly elegant way to do it is to frame such a translation as an institution morphism into an institution representing a specific process algebra, for example the one proposed by M. Roggenbach [30] for CSP.

Acknowledgements. The comments of the anonymous referees to earlier versions of this paper were fundamental to improve contents and exposition. This work was funded by ERDF European Regional Development Fund, through the

COMPETE Programme, and by National Funds through FCT - Portuguese Foundation for Science and Technology - within projects POCI-01-0145-FEDER-016692 (DaLí - Dynamic logics for cyber-physical systems: towards contract based design) and UID/MAT/04106/2013 at CIDMA. Further support was given by the project *SmartEGOV*, NORTE-01-0145-FEDER-000037, supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the EFDR. The first author is also supported by a FCT individual grant SFRH/BPD/103004/2014.

References

[1] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[2] C. Areces, P. Blackburn, and M. Marx. A road-map on complexity for hybrid logics. In J. Flum and M. Rodríguez-Artalejo, editors, *13th Intern. Workshop Computer Science Logic (CSL'99, Madrid, Spain, September 20-25, 1999)*, volume 1683 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 1999.

[3] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[4] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational theories of communicating processes*. Cambridge University Press, 2010.

[5] M. Bidoit and R. Hennicker. Constructor-based observational logic. *J. Log. Algebr. Program.*, 67(1-2):3–51, 2006.

[6] P. Blackburn and M. Tzakova. Hybrid languages and temporal logic. *Logic Journal of the IGPL*, 7(1):27, 1999.

[7] T. Braüner. *Hybrid Logic and its Proof-Theory*. App. Logic Series. Springer, 2010.

[8] M. V. Cengarle. The temporal logic institution. Technical Report Technical Report 9805, LUM München, Institut für Informatik, 1998.

[9] R. Diaconescu. Institutional semantics for many-valued logics. *Fuzzy Sets and Systems*, 218:32–52, 2013.

[10] J. L. Fiadeiro and T. S. E. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Asp. Comput.*, 4(3):239–272, 1992.

[11] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[12] R. Goldblatt. *Logics of Time and Computation*. Number 7 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA, 2 edition, 1992.

[13] V. Goranko. Temporal logic with reference pointers. In D. M. Gabbay and H. J. Ohlbach, editors, *First Int. Conf. Temporal Logic, ICTL*, volume 827 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 1994.

[14] R. Gorrieri, A. Rensink, and M. A. Zamboni. Action refinement. In *Handbook of Proacess Algebra*, pages 1047–1147. Elsevier, 2000.

[15] J. F. Groote and M. R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.

[16] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[17] K. Havelund. *The Fork Calculus -Towards a Logic for Concurrent ML*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994.

[18] R. Hennicker and A. Madeira. Behavioural semantics for the dynamic logic with binders. In M. Roggenbach and N. Oliet, editors, *Recent Trends in Algebraic Development Methods - Selected Papers of WADT 2016*. Springer, (in print).

[19] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[20] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[21] C. B. Jones. *Software Development — a Rigorous Approach*. Series in Computer Science. Prentice Hall, 1980.

[22] P. Knijnenburg and J. van Leeuwen. On models for propositional dynamic logic. *Theoretical Computer Science*, 91(2):181 – 203, 1991.

[23] K. G. Larsen and B. Thomsen. A modal process logic. In *Third Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society, 1988.

[24] A. Madeira, L. S. Barbosa, R. Hennicker, and M. A. Martins. Dynamic logic with binders and its application to the development of reactive systems. In A. Sampaio and F. Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Springer Lecture Notes in Computer Science*, pages 422–440, 2016.

[25] J. Magee and J. Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.

[26] L. O'Reilly, T. Mossakowski, and M. Roggenbach. Compositional modelling and reasoning in an institution for processes and data. In T. Mossakowski and H.-J. Kreowski, editors, *WADT 2010, Selected Papers*, pages 251–269. Springer, 2012.

[27] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science (5th GI-Conference)*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[28] G. Reggio, E. Astesiano, and C. Choppy. Casl-ltl: A casl extension for dynamic reactive systems version 1.0. – summary. technical report disi-tr-03-36. Technical report, DFKI Lab Bremen, 2013.

[29] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.

[30] M. Roggenbach. CSP-CASL - A new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354(1):42–71, 2006.

[31] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Inform.*, 25(3):233–281, 1988.

[32] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs on TCS, an EATCS Series. Springer, 2012.

[33] E. Sekerinski and K. Sere. *Program Development by Refinement: Case Studies Using the B Method*. Springer Science and Business Media. Springer, 2012.

[34] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 4)*, pages 1–148. Oxford University Press, Oxford, UK, 1995.